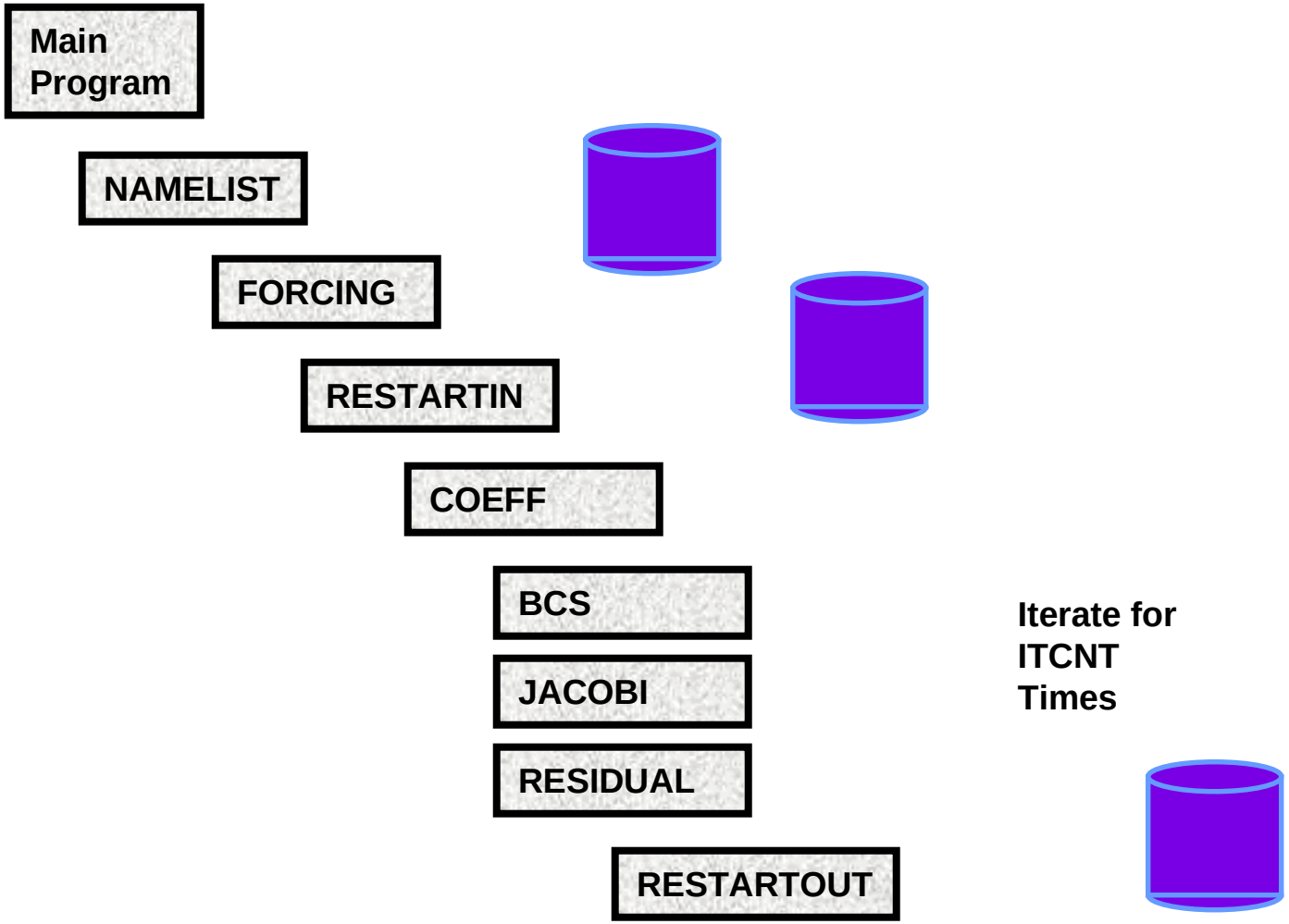
An Introduction to Input/Output Using MPI for the 1-D Decomposed Stommel Model

Luke Lonergan
High Performance Technologies, Inc.

Parallel I/O Exercise

Description of Problem
Overview of Alternative Approaches
Development of Basic Approach
Description of Scalable I/O Approach (Next Steps)
Performance Profiling Laboratory

Flowchart for the Serial Code



I/O Problem Description

Input and Output needs to be enabled in the Stommel MPI code

We will focus on the Forcing file I/O as our example case:

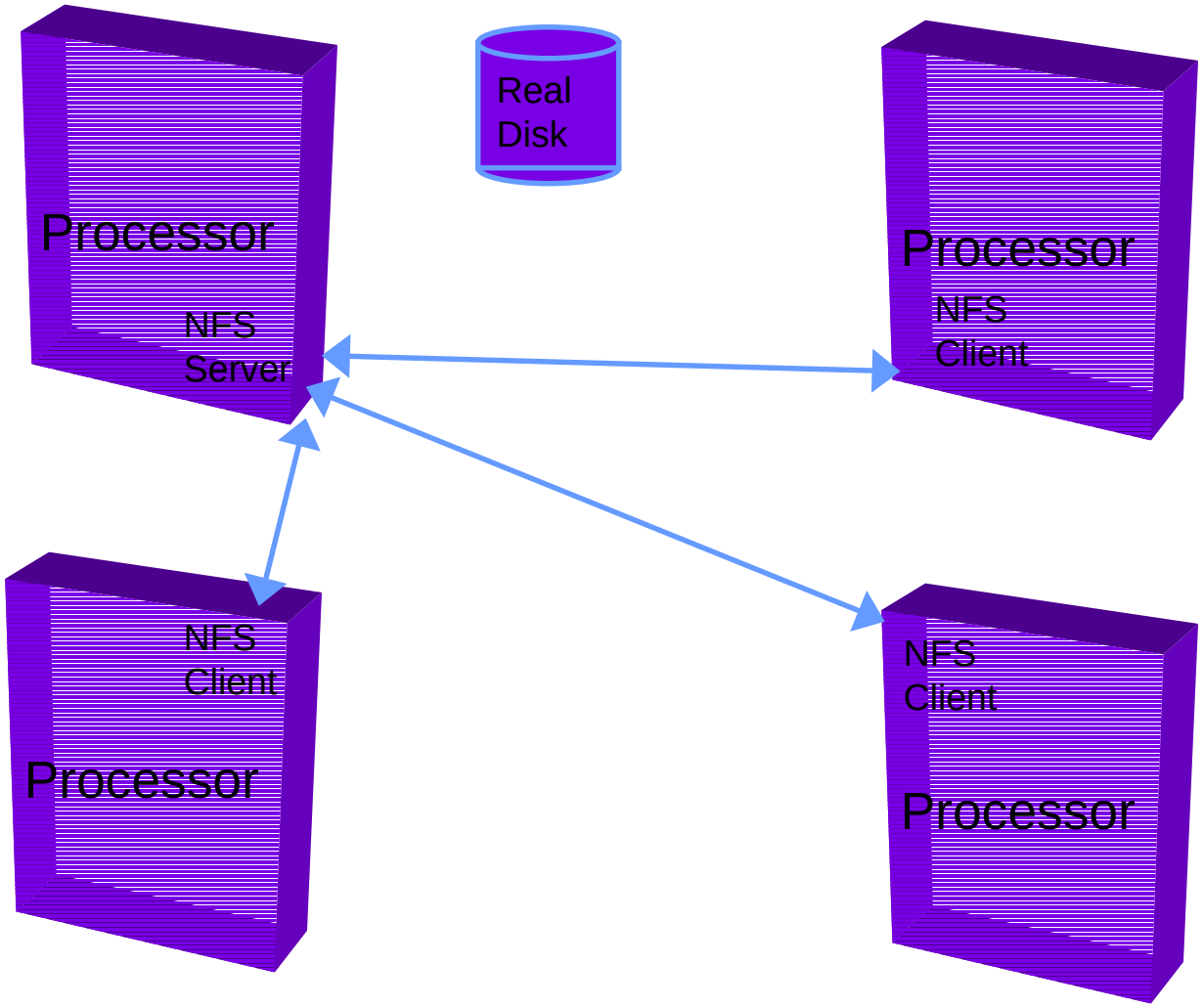
- May have been created from a serial run in our example code.
- Preferably created once, then read for subsequent runs.
- In a real oceanography code, came from another program or data source in a single file.

I/O Problem Description

Assumptions:

- Main program is sized too small to read the whole file
- There is a shared file system available to all processors, but it is located on one system

Shared File System

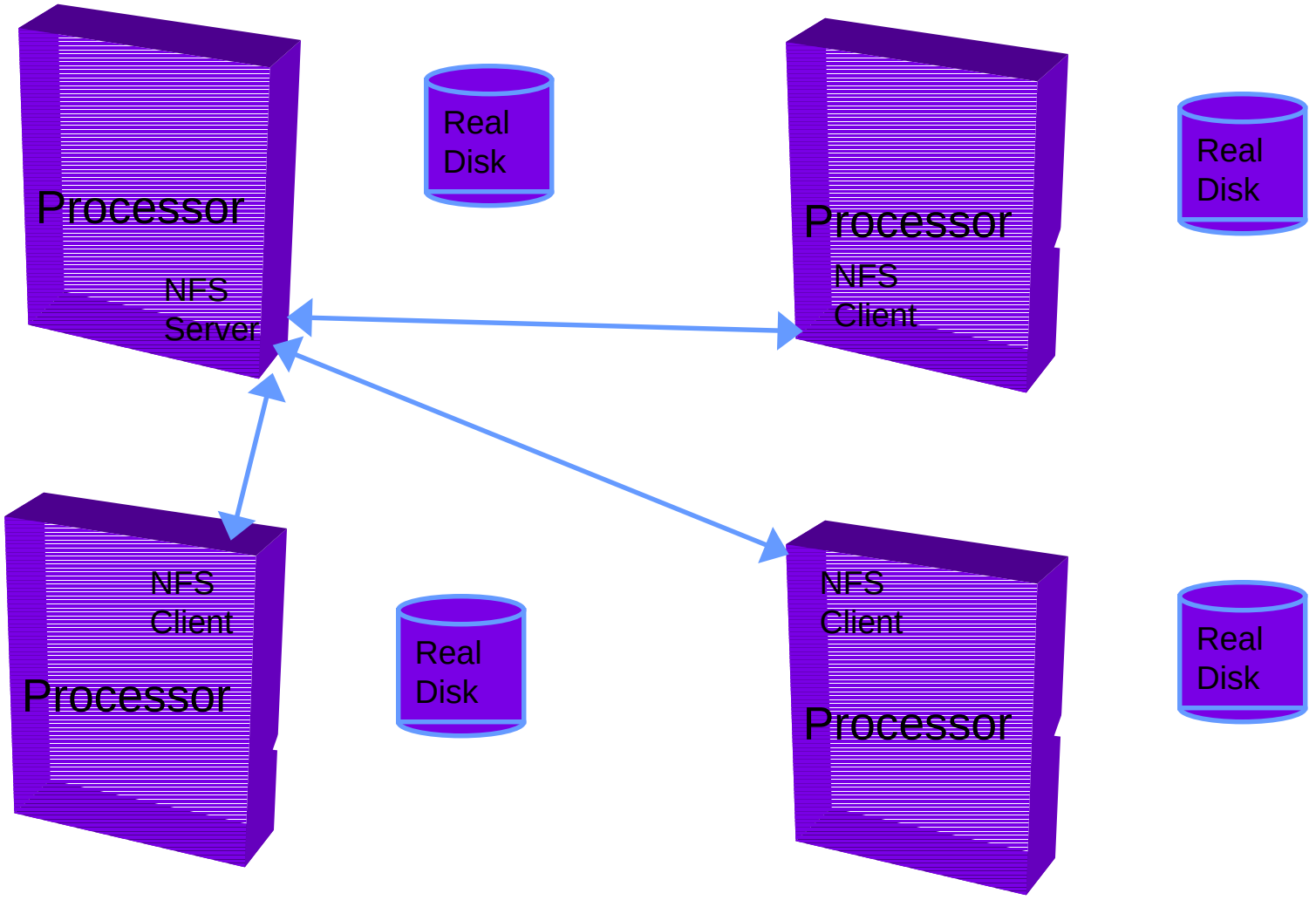


I/O Problem Description (Cont'd)

Depending on our requirements, we may need *scalable* I/O

- If we performed reads or writes to files every time step
- If the I/O at startup/shutdown is a significant fraction of execution time
- If the shared file system performance is poor
- ...

Distributed File System



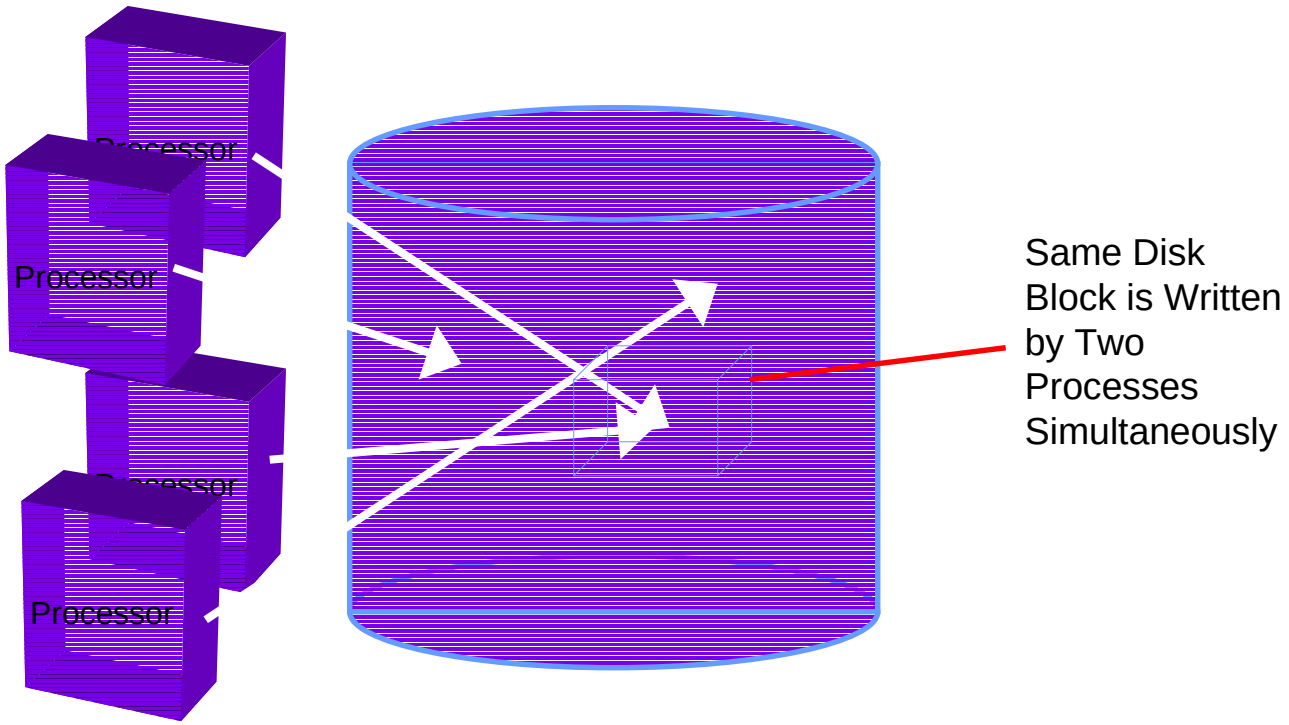
Distributed File System

If we take advantage of the distributed file system for scalable I/O, we need to distribute subdivided pieces of the file to the separate disk subsystems

We now have the problem of files not necessarily being on the same processor where they are needed

Simultaneous I/O

If our N copies of the Stommel program are reading and writing the same file at the same time, they will "bump heads" with unpredictable results for writes:



Alternative Approaches

Approach A: (Scalable [with 2N possible messages] but hard)

If the file named with the type.{rank#} designation does not exist on the local processor, generate it there on the local filesystem, if possible.

If it is not possible to generate the file, segment a pre-existing file into pieces and copy the right piece to the local filesystem.

Approach A (Cont’d)

If a file does exist with type.{rank#}, read it. If the {rank#} does not match the rank of the process reading it, initiate a message to the appropriate rank with the contents of the forcing file.
Wait for a message from any other rank with the contents of this rank's file.

Approach B

Approach B: (Scalable and easier than A, with 2N less messages)

If the file named with the type.{rank#} designation does not exist on the local processor, generate it there on the local filesystem, if possible. (Same as A)

If it is not possible to generate the file, segment a pre-existing file into pieces and copy the right piece to the local filesystem. (Same as A)

Approach B (Cont’d)

If a file does exist with type.{rank#}, read it. Create a new communicator that establishes this processor's rank as the same as that of the file it has read.

Approach C

Approach C: (Not Scalable, but relatively simple and High Performance)

If the file named with the type.{rank#} designation does not exist on the _shared_ filesystem, generate it there. If it is not possible to generate the file, segment a pre-existing file into pieces in place on the shared filesystem.

If a file does exist with type.{_my_rank#}, read it.

Create a New Format Forcing File

We select a new file format due to the need to decompose the forcing function array into pieces, one for each file segment

Decomposition and Reconstruction are made simpler as a result

Create a New Format Forcing File

Change directory to mpi_1d_io/step1
Edit the README file
Recommended Approach:

- Look through the main program
- Type “make” to compile and link an executable
- Run the program to create a forcing file with the new format

Chop the Forcing File into Pieces

Process is called segmentation

This technique is the most generalizable approach for parallel I/O

If you can do I/O this way, you can do it on any distributed memory parallel machine

The process in general:

- Read in the whole file (not necessarily required, but we'll do it this way)
- Write out one piece of the file at a time into a unique file name, in this case derived from the original name

Decomposing the Forcing Function

$y(j) = (j-1) * \Delta y$

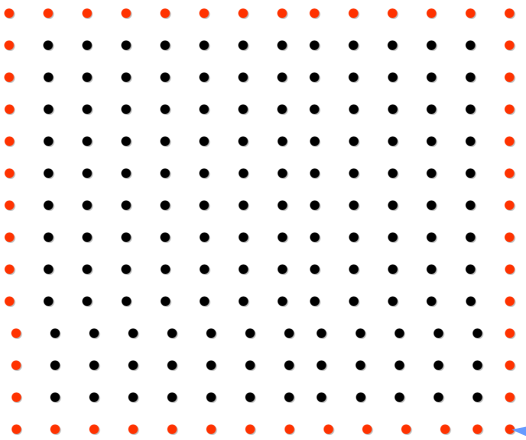
$j(y) = y/\Delta y + 1$

$y(j) = ystart + (j-1) * \Delta y$

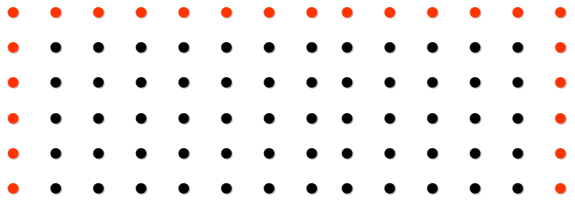
$ystart = rank * (myny-2) * \Delta y$

$jstart = rank * (myny-2) + 1$

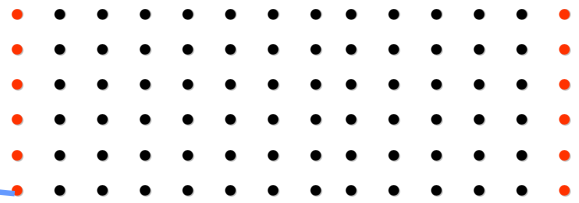
Jtotal = 14, Δy = 1, Ytotal = 13



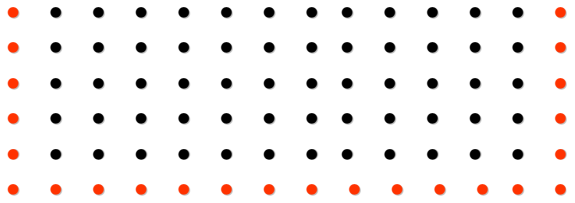
Nprocs = 3, MyNY = 6



Rank = 1, Ystart = 8, Jstart = 9



Rank = 1, Ystart = 4, Jstart = 5



Rank = 0, Ystart = 0, Jstart = 1

Chop the Forcing File into Pieces

Change directory to mpi_1d_io/step2

Edit the README file

Recommended Approach:

- Look through the main program
- Type “make” to compile and link an executable
- Copy the forcing file we created in step1 to the local directory, I.e. “cp ../step1/force.in .”
- Run the program to chop up force.in into force.in.[0,1,2,3,…, Nprocs]

Chop the Forcing File into Pieces

Experiment with different number of processors to chop the file into

Remember, the problem is only divisible by certain numbers, namely those that evenly divide (NJ-2), I.e. NPROC is valid if:

mod((NJ-2),NPROC) = 0

Re-Assemble the Forcing File

Change directory to mpi_1d_io/step3
Edit the README file
Recommended Approach:

- Look through the main program
- Type “make” to compile and link an executable
- Copy the chopped forcing file we created in step2 to the local directory, I.e. “cp ../step2/force.in.* .”
- Run the program to put force.in back together again

Run Stommel mpi1d with I/O!

Change directory to mpi_1d_io/step4

Edit the README file

Recommended Approach:

- Look through the main program and forcing.f
- Type “make” to compile and link an executable
- Run the complete MPI stommel code with the number of compiled processors
  - This will use the NFS filesystem

Run Stommel mpi1d with I/O!

Edit forcing.f and change the name of the forcing file base from “force.in” to “/tmp/my_initials_force.in”

Re-compile by typing ‘make’

Run the code with the compiled number of processors

- This will use the distributed filesystems of the processors you are using!
- Note the difference in runtime and Mflops